TensorLog: Deep Learning Meets Probabilistic Databases

William W. Cohen Fan Yang Kathryn Rivard Mazaitis
Machine Learning Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh PA 15208

Abstract

We present an implementation of a probabilistic first-order logic called TensorLog, in which classes of logical queries are compiled into differentiable functions in a neural-network infrastructure such as Tensorflow or Theano. This leads to a close integration of probabilistic logical reasoning with deep-learning infrastructure: in particular, it enables high-performance deep learning frameworks to be used for tuning the parameters of a probabilistic logic. Experimental results show that TensorLog scales to problems involving hundreds of thousands of knowledge-base triples and tens of thousands of examples.

1. Introduction

1.1 Motivation

Recent progress in deep learning has profoundly affected many areas of artificial intelligence. One exception is probabilistic first-order logical reasoning. In this paper, we seek to closely integrate probabilistic logical reasoning with the powerful infrastructure that has been developed for deep learning. The end goal is to enable deep learners to incorporate first-order probabilistic KBs, and conversely, to enable probabilistic reasoning over the outputs of deep learners.

As motivation, consider the program of Figure 1, which could be plausibly used for answering simple natural-language questions against a KB, such as "Who was the director of Apocalyse Now?" The main predicate `answer` takes a question and produces an answer (which would be an entity in the KB). The predicates `actedIn`, `directed`, etc, are from the KB. For the purpose of performing natural-language analysis, the KB has also been extended with facts about the text that composes the training and test data: the KB stores information about word n-grams contained in the question, the strings that are possible names of an entity, and the words that are contained in these names and n-grams. The underlined predicates `indicatesLabel`, `important`, and `popular` are "soft" KB predicates, and the goal of learning is to find appropriate weights for the soft-predicate facts—e.g., to learn that `indicatesLabel(director, aboutDirected)` has high weight. Ideally these weights would be learned indirectly, from observing inferences made using the KB. In this case we would like to learn from question-answer pairs, which rely indirectly on the KB predicates like `actedIn`, etc, rather than from hand-classified questions, or judgements about specific facts in the soft predicates.

TensorLog, the system we describe here, makes this possible to do at reasonable scale using conventional neural-network platforms. For instance, for a variant of the problem

```
answer(Question,Answer) :-
    classification(Question,aboutActedIn),
    mentionsEntity(Question,Entity), actedIn(Answer,Entity).
answer(Question,Answer) :-
    classification(Question,aboutDirected),
    mentionsEntity(Question,Entity), directed(Answer,Entity).
answer(Question,Answer) :-
    classification(Question,aboutProduced),
    mentionsEntity(Question,Entity), produced(Answer,Entity).
...
mentionsEntity(Question,Entity) :-
    containsNGram(Question,NGram), matches(NGram,Name),
    possibleName(Entity,Name), popular(Entity).

classification(Question,Y) :-
    containsNGram(Question,NGram), indicatesLabel(NGram,Y).
matches(NGram,Name) :-
    containsWord(NGram,Word), containsWord(Name,Word), important(Word).
```

Figure 1: A simple theory for question-answering against a KB.

above, we can learn from 10,000 questions against a KB of 420,000 tuples in around 200 seconds per epoch, on a typical desktop with a single GPU.

1.2 Approach and Contributions

The main technical obstacle to integration of probabilistic logics into deep learners is that most existing first-order probabilistic logics are not easily adapted to evaluation on a GPU. One superficial problem is that the computations made in theorem-proving are not numeric, but there is also a more fundamental problem, which we will now discuss.

The most common approach to first-order inference is to "ground" a first-order logic by converting it to a zeroth-order format, such as a boolean formula or a probabilistic graphical model. For instance, in the context of a particular KB, the rule

$$p(X,Y) \leftarrow q(Y,Z), r(Z,Y). \tag{1}$$

can be "grounded" as the following finite boolean disjunction, where $\mathcal{C}$ is the set of objects in the KB:

$$\bigvee_{\exists x,y,z\in\mathcal{C}} (p(x,y) \vee \neg q(y,z) \vee \neg r(z,y))$$

This boolean disjunction can embedded in a neural network, e.g. to initialize an architecture (Towell, Shavlik, & Noordewier, 1990) or as a regularizer (Hu, Ma, Liu, Hovy, & Xing, 2016; Rocktschel, Singh, & Riedel, 2015). For probabilistic first-order languages (e.g., Markov logic networks (Richardson & Domingos, 2006)), grounding typically results in a directed graphical model (see (Kimmig, Mihalkova, & Getoor, 2015) for a survey of this work).

The problem with this approach is that groundings can be very large: even the small rule above gives a grounding of size $o(|\mathcal{C}|^3)$, which is likely much larger than the size of the KB, and a grounding of size $o(|\mathcal{C}|^n)$ is produced by a rule like

$$p(X_0, X_n) \leftarrow q_1(X_0, X_1), q_2(X_1, X_2), \ldots, q_n(X_{n-1}, X_n) \qquad (2)$$

The target architecture for modern deep learners is based on GPUs, which have limited memory: hence the grounding approach can be used only for small KBs and short rules. For example, (Serafini & Garcez, 2016) describes experimental results with five rules and a few dozen facts, and the largest datasets considered by (Sourek, Aschenbrenner, Zelezný, & Kuzelka, 2015) contain only about 3500 examples.

Although not all probabilistic logic implementations require explicit grounding, a similar problem arises in using neural-network platforms to implement any probabilistic logic which is computationally hard. For many probabilistic logics, answering queries is #P-complete or worse. Since the networks constructed in modern deep learning platforms can be evaluated in time polynomial in their size, no polysize network can implement such a logic, unless #P=P.

This paper addresses these obstacles with several interrelated contributions. First, in Section 2, we identify a restricted family of probabilistic deductive databases (PrDDBs) called *polytree-limited stochastic deductive knowledge graphs (ptree-SDKGs)* which are tractable, but still reasonably expressive. This formalism is a variant of stochastic logic programs (SLPs). We also show that ptree-SDKGs are in some sense maximally expressive, in that we cannot drop the polytree restriction, or switch to a more conventional possible-worlds semantics, without making inference intractible.

Next, in Section 3, we present an algorithm for performing inference for ptree-SDKGs. This algorithm performs inference with a dynamic-programming method, which we formalize as belief propagation on a certain factor graph, where each random variable in the factor graph correspond to possible bindings to a logical variable in a proof, and the factors correspond to database predicates. In other words, the random variables are multinomials over all constants in the database, and the factors constrain these bindings to be consistent with database predicates that relate the corresponding logical variables. Although this is a simple idea, to our knowledge it is novel. We also discuss in some detail our implementation of this logic, called TensorLog.

We finally discuss related work, experimental results, and present conclusions.

2. Background

2.1 Deductive DBs

In this section we review the usual definitions for logic programs and deductive databases, and also introduce the term *deductive knowledge graph (DKG)* for deductive databases containing only unary and binary predicates. This section can be omitted by readers familiar with logic programming.

An example of a *deductive database* (DDB) is shown in Figure 2. A *database*, $\mathcal{DB}$, is a set $\{f_1, \ldots, f_N\}$ of ground facts. (For the moment, ignore the numbers associated with each database fact in the figure.) A theory, $\mathcal{T}$, is a set of function-free Horn clauses. Clauses are

```
1. uncle(X,Y):-child(X,W),brother(W,Y).
2. uncle(X,Y):-aunt(X,W),husband(W,Y).
3. status(X,tired):-child(W,X),infant(W).
```

`child(liam,eve)`	0.99
`child(dave,eve)`	0.99
`child(liam,bob)`	0.75
`husband(eve,bob)`	0.9
`infant(liam)`	0.7
`infant(dave)`	0.1
`aunt(joe,eve)`	0.9
`brother(eve,chip)`	0.9

Figure 2: An example database and theory. Uppercase symbols are universally quantified variables, and so clause 3 should be read as a logical implication: for all database constants c_X and c_W, if `child(`c_X`,`c_W`)` and `infant(`c_W`)` can be proved, then `status(`c_X`,tired)` can also be proved.

$$\begin{tikzcd}
 & \text{(S=uncle(liam,Y), L=[uncle(liam,Y)])} \arrow[d] & \\
 & \text{(S=uncle(liam,Y), L=[child(liam,W),brother(W,Y)])} \arrow[dl] \arrow[dr] & \\
\text{(S=uncle(liam,Y), L=[brother(bob,Y)])} \arrow[d] & & \text{(S=uncle(liam,Y), L=[brother(eve,Y)])} \arrow[d] \\
\textit{dead end} & & \text{(S=uncle(liam,chip), L=[])}
\end{tikzcd}$$

Figure 3: An example proof tree. From root to second level uses rule 1; next level uses unit clause `child(liam,bob):-` on left and unit clause `child(liam,eve):-` on right; final level uses `brother(eve,chip):-` on the right.

written A`:-`$B_1, \ldots, B_k$, where A is called the *head* of the clause, $B_1, \ldots, B_k$ is the *body*, and A and the B_i's are called *literals*. Literals must be of the form $p(X_1, \ldots, X_k)$, where p is a *predicate symbol* and the X_i's either logical variables or database constants. The set of all database constants is written $\mathcal{C}$. The number of arguments k to a literal is called its *arity*.

In this paper we focus on the case where all literals are binary or unary, i.e., have arity no more than two. We will call such a database a *knowledge graph (KG)*, and the program a *deductive knowledge graph (DKG)*. We will also assume that constants appear only in the database, not in the theory (although this assumption can be relaxed).

Clauses can be understood as logical implications. Let σ be a *substitution*, i.e., a mapping from logical variables to constants in $\mathcal{C}$, and let $\sigma(L)$ be the result of replacing all logical variables X in the literal L with $\sigma(X)$. A set of tuples S is *deductively closed* with respect to the clause $A \leftarrow B_1, \ldots, B_k$ iff for all substitutions σ, either $\sigma(A) \in S$ or $\exists B_i : \sigma(B_i) \notin S$. For example, if S contains the facts of Figure 2, S is not deductively closed with respect to the clause 1 unless it also contains `uncle(chip,liam)` and `uncle(chip,dave)`. The *least model* for a pair $\mathcal{DB}, \mathcal{T}$, written *Model*$(\mathcal{DB}, \mathcal{T})$, is the smallest superset of $\mathcal{DB}$ that is deductively closed with respect to every clause in $\mathcal{T}$. This least model is unique, and in the usual DDB semantics, a ground fact f is considered "true" iff $f \in$ *Model*$(\mathcal{DB}, T)$.

There are two broad classes of algorithms for inference in a DDB. *Bottom-up inference* explicitly computes the set *Model*$(\mathcal{DB}, \mathcal{T})$ iteratively. Bottom-up inference repeatedly extends a set of facts S, which initially contains just the database facts, by looking for rules which "fire" on S and using them derive new facts. (More formally, one looks for rules $A \leftarrow B_1, \ldots, B_k$ and substitutions σ such that $\forall i$, $\sigma(B_i) \in S$, and then adds the derived fact $\sigma(A)$ to S.) This process is then repeated until it converges. For DDB programs, bottom-up inference takes time polynomial in the size of the database $|\mathcal{DB}|$, but exponential in the length of the longest clause in $\mathcal{T}$ (Ramakrishnan & Ullman, 1995).

One problem with bottom-up theorem-proving is that it explicitly generates *Model*$(\mathcal{DB}, \mathcal{T})$, which can be much larger than the original database. The alternative is *top-down inference*. Here, the algorithm does not compute a least model explicitly: instead, it takes as input a query fact f and determines whether f is derivable, i.e., if $f \in$ *Model*$(\mathcal{DB}, \mathcal{T})$. More generally, one might retrieve all derivable facts that match some pattern, e.g., find all values of Y such that `uncle(joe,Y)` holds. (Formally, given $Q =$ `uncle(joe,Y)`, we would like to find all $f \in$ *Model*$(\mathcal{DB}, \mathcal{T})$ which are *instances of* Q, where an f is defined to be an *instance of* Q iff $\exists \sigma : f = \sigma(Q)$..

To describe top-down theorem-proving, we note that facts in the database can also be viewed as clauses: in particular a fact $p(a, b)$ can be viewed as a clause $p(a, b) \leftarrow$ which has $p(a, b)$ as its head and an empty body. This sort of clause is called a *unit clause*. We will use $\mathcal{T}^{+\mathcal{DB}}$ to denote the theory $\mathcal{T}$ augmented with unit clauses for each database fact. A top-down theorem prover can be viewed as constructing and searching a following tree, using the theory $\mathcal{T}^{+\mathcal{DB}}$. The process is illustrated in Figure 3, and detailed below.

1. The root vertex is a pair (S, L), where S is the query Q, and L is a list containing only Q. In general every vertex is a pair where S is something derived from Q, and L is a list of literals left to prove.

2. For any vertex (S, L), where $L = [G_1, \ldots, G_n]$, there is a child vertex (S', L') for each rule $A \leftarrow B_1, \ldots, B_k \in \mathcal{T}^{+\mathcal{DB}}$ and substitution σ for which $\sigma(G_i) = \sigma(A)$ for some

G_i. In this child node, $S' = \sigma(S)$, and

$$L' = [\sigma(G_1), \ldots, \sigma(G_{i-1}), \sigma(B_1), \ldots, \sigma(B_k), \sigma(G_{i+1}), \ldots, \sigma(G_n)]$$

Note that L' is smaller than L if the clause selected is a unit clause (i.e., a fact). If L' is empty, then the vertex is called a *solution vertex*. In any solution vertex (S, L), if S contains no variables,[1] then S is an instance of Q and is in $\mathit{Model}(\mathcal{T}, \mathcal{DB})$.

If $\mathcal{T}$ is not recursive, or if recursion is limited to a fixed depth, then the proof graph is finite. We will restrict our discussion below to theories with finite proof graphs. For this case, the set of all answers to a query Q can be found by systematically searching the proof tree for all solution vertices. A number of strategies exist for this, but one popular one is that used by Prolog, which uses depth-first search, ordering edges by picking the first rule $A \leftarrow B_1, \ldots, B_k$ in a fixed order, and only matching rules against the first element of L. This strategy can be implemented quite efficiently and is easily extended to much more general logic programs.

2.2 SLPs and stochastic deductive KGs

There are a number of approaches to incorporating probabilistic reasoning in first-order logics. We focus here on *stochastic logic programs (SLPs)* (Cussens, 2001), in which the theory $\mathcal{T}$ is extended by associating with each rule r a non-negative scalar weight θ_r. Below we summarize the semantics associated with SLPs, for completeness, and refer the reader to (Cussens, 2001) for details.

In an SLP weights θ_r are added to edges of the top-down proof graph the natural way: when a rule r is used to create an edge $(S, L) \rightarrow (S', L')'$, this edge is given weight θ_r. We define the weight of a path $v_1 \rightarrow \ldots \rightarrow v_n$ in the proof graph for Q to be the product of the weights of the edges in the path, and the weight of a node v to be the sum of the weights of the paths from the root note $v_0 = (Q, [Q])$ to v. If $r_{v,v'}$ is the rule used for the edge from v to v', then the weight of $w_Q(v_n)$ is

$$w_Q(v_n) \equiv \sum_{v_0 \rightarrow \ldots \rightarrow v_n} \prod_{i=0}^{n-1} \theta_{r_{v_i, v_{i+1}}}$$

The weight of an answer f to query Q is defined by summing over paths to solution nodes that yield f:

$$w_Q(f) \equiv \sum_{v : v = (f, [])} w_Q(v) \tag{3}$$

(Here [] is the empty list, which indicates a solution vertex has been reached.) Finally, if we assume that some answers to Q do exist, we can produce a conditional probability distribution over answers f to the query Q by normalizing w_Q, i.e.,

$$\Pr(f|Q) \equiv \frac{1}{Z} w_Q(f)$$

1. If S does have variables in it, then any fact f which can be constructed by replacing variables in Q with database constants is in the least model. For clarity we will ignore this complication in the discussion below.

Following the terminology of (Cussens, 2001) this is a *pure unnormalized SLP*. SLPs were originally defined (Muggleton et al., 1996) for a fairly expressive class of logic programs, namely all programs which are *fail free*, in the sense that there are no "dead ends" in the proof graph (i.e., from every vertex v, at least one solution node is reachable). Prior work with SLPs also considered the special case of *normalized SLPs*, in which the weights of all outgoing edges from every vertex v sum to one. For normalized fail-free SLPs, it is simple to modify the usual top-down theorem prover to sample from $Pr(f|Q)$.

2.3 Stochastic deductive KGs and discussion of SLPs

SLPs are closely connected to several other well-known types of probabilistic reasoners. SLPs are defined by introducing probabilistic choices into a top-down theorem-proving process: since top-down theorem-proving for logic programs is analogous to program execution in ordinary programs, SLPs can be thought of as logic-program analogs to probabilistic programming languages like Church (Goodman, Mansinghka, Roy, Bonawitz, & Tenenbaum, 2012). Normalized SLPs are also conceptually quite similar to stochastic grammars, such as pCFGs, except that stochastic choices are made during theorem-proving, rather than rewriting a string.

Here we consider three restrictions on SLPs. First, we restrict the program to be in DDB form—i.e., it consists of a theory $\mathcal{T}$ which contains function-free clauses, and a database $\mathcal{DB}$ (of unit clauses). Second, we restrict all predicates to be unary or binary. Third, we restrict the clauses in the theory $\mathcal{T}$ to have weight 1, so that the only meaningful weights are associated with database facts. We call this restricted SLP a *stochastic deductive knowledge graph (SDKG)*.

For SDKGs, a final connection with other logics can be made by considering a logic program that has been grounded by conversion to a boolean formulae. One simple approach to implementing a "soft" extension of a boolean logic is to evaluate the truth or falsity of a formula bottom-up, deriving a numeric confidence c for each subexpression from the confidences associated with its subparts. For instance, one might use the rules

$$\begin{aligned}
c(x \wedge y) &\equiv \min(c(x), c(y)) \\
c(x \vee y) &\equiv \max(c(x), c(y)) \\
c(\neg x) &\equiv 1 - c(x)
\end{aligned}$$

This approach to implementing a soft logic is is sometimes called an *extensional* approach (Suciu, Olteanu, Ré, & Koch, 2011), and it is common in practical systems: PSL (Brocheler, Mihalkova, & Getoor, 2010) uses an extensional approach, as do several recent neural approaches (Serafini & Garcez, 2016; Hu et al., 2016).

Now consider modifying a top-down prover to produce a particular boolean formula, in which each path $v_0 \rightarrow \ldots \rightarrow v_n$ is associated with a conjunction $f_1 \wedge \ldots \wedge f_m$ of all unit-clause facts used along this path, and each answer f is associated with the disjunction of these conjunctions. Then let us compute the unnormalized weight $w_Q(f)$ using the rules

$$\begin{aligned}
c(x \wedge y) &\equiv c(x) \cdot c(y) \\
c(x \vee y) &\equiv c(x) + c(y)
\end{aligned}$$

(which are sufficient since no negation occurs in the formula). This (followed by normalization) can be shown to be equivalent to the SLP semantics.

2.4 Complexity of reasoning with stochastic deductive KGs

SLPs have a relatively simple proof procedure: informally, inference only requires computing a weighted count of all proofs for a query, and the weight for any particular proof can be computed quickly. A natural question is whether computationally efficient theorem-proving schemes exist for SLPs. The similarity between SLPs and probabilistic context-free grammars suggests that efficient schemes might exist, since there are efficient dynamic-programming methods for probabilistic parsing. Unfortunately, this is not the case: even for the restricted case of SDKGs, computing $P(f|Q)$ is #P-hard.

Theorem 1 *Computing $P(f|Q)$ (relative to a SDKG $\mathcal{T}, \mathcal{DB}$) for all possible answers f of the query Q is #P-hard, even if there are only two such answers, the theory contains only two non-recursive clauses, and the KG contains only 13 facts.*

A proof appears in the appendix. The result is not especially surprising, as it is easy to find small theories with exponentially many proofs: e.g., the clause of Equation2 can have exponentially many proofs, and naive proof-counting methods may be expensive on such a clause.

Fortunately, one further restriction makes SLP theorem-proving efficient.

For a theory clause $r = A \leftarrow B_1, \ldots, B_k$, define the *literal influence graph for r* to be a graph where each B_i is a vertex, and there is an edge from B_i to B_j iff they share a variable. A graph is a *polytree* iff there is at most one path between any pair of vertices: i.e., if each strongly connected component of the graph is a tree. Finally, we define a theory to be *polytree-limited* iff the influence graph for every clause is a polytree. Figure 4 contains some examples of polytree-limited clauses.

This additional restriction makes inference tractable.

Theorem 2 *For any SDKG with a non-recursive polytree-limited theory $\mathcal{T}$, $P(f|Q)$ can be computed in computed in time linear in the size of $\mathcal{T}$ and $\mathcal{DB}$.*

The proof follows from the correctness of a dynamic-programming algorithm for SDKG inference, which we will present below, in detail, in Section 3. In brief, the algorithm is based on belief propagation in a certain factor graph. We construct a graph where the random variables are multinomials over the set of all database constants, and each random variable corresponds to a logical variable in the proof graph. The logical literals in a proof correspond to factors, which constrain the bindings of the variables to make the literals true.

Importantly for the goal of compilation into deep-learning frameworks, the message-passing steps used for belief propagation can be defined as numerical operations, and given a predicate and an input/output mode, the message-passing steps required to perform belief propagation (and hence inference) can be "unrolled" into a function, which is differentiable.

2.5 Complexity of stochastic DKGs variants

2.5.1 Extensions that maintain efficiency

Constants in the theory. We will assume that constants appear only in the database, not in the theory. To relax this, note that it is possible to introduce a constant into a theory by creating a special unary predicate which holds only for that constant: e.g., to use the constant `tired`, one could create a database predicate `assign_tired(T)` which contains the one fact `assign_tired(tired)`, and use it to introduce a variable which is bound to the constant `tired` when needed. For instance, the clause 3 of Figure 2 would be rewritten as

```
status(X,T):-assign_tired(T),child(X,W),infant(W).    (4)
```

Without loss of generality, we assume henceforth that constants only appear in literals of this sort.

Rule weights and rule features. In a SDKG, weights are associated only with *facts* in the databases, not with *rules* in the theory (which differs from the usual SLP definition). However, there is a standard "trick" which can be used to lift weights from a database into rules: one simply introduces a special clause-specific fact, and add it to the clause body (Poole, 1997). For example, a weighted version of clause 3 could be re-written as

```
status(X,tired):-assign_c3(RuleId),weighted(RuleId),child(W,X),infant(W)
```

where the (parameterized) fact `weighted(c3)` appears in $\mathcal{DB}$, and the constant `c3` appears nowhere else in $\mathcal{DB}$.

In some probabilistic logics, e.g., ProPPR (Wang, Mazaitis, & Cohen, 2013) one can attach a computed set of features to a rule in order to weight it: e.g., one can write

```
status(X,tired):-{weighted(A):child(W,X),age(W,A)}
```

which indicates that the all the ages of the children of X should be used as features to determine if the rule succeeds. This is equivalent to the rule `status(X,tired) :-child(W,X), age(W,A), weighted(A)`, and in the experiments below, where we compare to ProPPR, we use this construction.

2.5.2 Extension to possible-worlds semantics

In the SLP semantics, the parameters Θ only have meaning in the context of the set of proofs derivable using the theory $\mathcal{T}$. This can be thought of as a "possible proofs" semantics. It has been argued that it is more natural to adopt a "possible worlds" semantics, in which Θ is used to define a distribution, $\Pr(I|\mathcal{DB},\Theta)$, over "hard" databases, and the probability of a derived fact f is defined as follows, where $[\![\cdot]\!]$ is a zero-one indicator function:

$$\Pr_{\texttt{TupInd}}(f|\mathcal{T},\mathcal{DB},\Theta) \equiv \sum_I [\![f \in \mathit{Model}(I,\mathcal{T})]\!] \cdot \Pr(I|\mathcal{DB},\Theta) \quad (5)$$

Potential hard databases are often called *interpretations* in this setting. The simplest such "possible worlds" model is the *tuple independence* model for PrDDB's (Suciu et al., 2011): in this model, to generate an interpretation I, each fact $f \in \mathcal{DB}$ sampled by independent coin tosses, i.e., $\Pr_{\texttt{TupInd}}(I|\mathcal{DB},\Theta) \equiv \prod_{t\in I}\theta_t \cdot \prod_{t\in\mathcal{DB}-I}(1-\theta_t)$.

ProbLog (Fierens, Broeck, Renkens, Shterionov, Gutmann, Thon, Janssens, & Raedt, 2016) is one well-known logic programming language which adopts this semantics, and there is a large literature (for surveys, see (Suciu et al., 2011; De Raedt & Kersting, 2008)) on approaches to more tractibly estimating Eq 5, which naively requires marginalizing over all $2^{|\mathcal{DB}|}$ interpretations. A natural question to ask is whether polytree-limited SDKGs, which are tractible under the possible-proofs semantics of SLPs, are also tractible under a possible-worlds semantics. Unfortunately, this is not the case.

Theorem 3 *Computing $P(f)$ in the tuple-independent possible-worlds semantics for a single ground fact f is #P-hard.*

This result is well known: for instance, Suciu and Olteanu (Suciu et al., 2011) show that it is #P-hard to compute probabilities against the one-rule theory `p(X,Y) :- q(X,Z),r(Z,Y).` For completeness, the appendix to this paper contains a proof, which emphasizes the fact that reasonable syntactic restrictions (such as polytree-limited theories) are unlikely to make inference tractible. In particular, the theory used in the construction is extremely simple: all predicates are unary, and contain only three literals in their body.

3. Efficient differentiable inference for polytree-limited SDKGs

In this section we present an efficient dynamic-programming method for inference in polytree-limited SDKGs. We formalize this method as belief propagation on a certain factor graph, where the random variables in the factor graph correspond to possible bindings to a logical variable in a proof, and the factors correspond to database predicates. In other words, the random variables are multinomials over all constants in the database, and the factors will constrain these bindings to be consistent with database predicates that related the corresponding logical variables.

Although using belief propagation in this way is a simple idea, to our knowledge it is a novel method for first-order probabilistic inference. Certainly it is quite different from more common formulations of first-order probabilistic inference, where random variables typically are Bernoulli random variables, which correspond to *potential* ground database facts (i.e., elements of the Herbrand base of the program.)

3.1 Numeric encoding of PrDDB's and queries

Because our ultimate goal is integration with neural networks, we will implement reasoning by defining a series of numeric functions, each of which finds answers to a particular family of queries. It will be convenient to encode the database numerically. We will assume all constants have been mapped to integers. For a constant $c \in \mathcal{C}$, we define $\mathbf{u}_c$ to be a one-hot row-vector representation for c, i.e., a row vector of dimension $|\mathcal{C}|$ where $\mathbf{u}[c] = 1$ and $\mathbf{u}[c'] = 0$ for $c' \neq C$. We can also represent a binary predicate p by a sparse matrix $\mathbf{M}_p$, where $\mathbf{M}_p[a, b] = \theta_{p(a,b)}$ if $p(a, b) \in \mathcal{DB}$, and a unary predicate q as an analogous row vector $\mathbf{v}_q$. Note that $\mathbf{M}_p$ encodes information not only about the database facts in predicate p, but also about their parameter values. Collectively, the matrices M_{p_1}, ..., M_{p_n} for the predicates $p_1, \ldots, p_n$ can be viewed as a three-dimensional tensor.

Our main interest here is queries that retrieve all derivable facts that match some query Q: e.g., to find all values of Y such that `uncle(joe,Y)` holds. We define an *argument-retrieval*

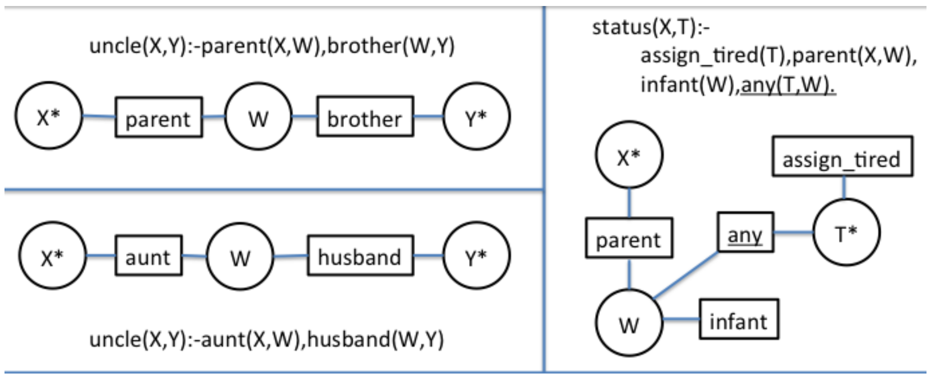
uncle(X,Y):-parent(X,W),brother(W,Y)
X*
parent
W
brother
Y*
X*
aunt
W
husband
Y*
uncle(X,Y):-aunt(X,W),husband(W,Y)
status(X,T):-
assign_tired(T),parent(X,W),
infant(W),any(T,W).
X*
assign_tired
parent
any
T*
W
infant

Figure 4: Examples of factor graphs for the example theory.

query Q as query of the form $p(c,Y)$ or $p(Y,c)$. We say that $p(c,Y)$ has an *input-output mode* of `in,out` and $p(Y,c)$ has an input-output mode of `out,in`. For the sake of brevity, below we will assume below the mode `in,out` when possible, and abbreviate the two modes as `io` and `io`.

The *response* to a query $p(c,Y)$ is a distribution over possible substitutions for Y, encoded as a vector $\mathbf{v}_Y$ such that for all constants $d \in \mathcal{C}$, $\mathbf{v}_Y[d] = \Pr(p(c,d)|Q = p(x,Y), \mathcal{T}, \mathcal{DB}, \Theta)$. Note that in the SLP model $\mathbf{v}_Y$ is a conditional probability vector, conditioned of $Q = p(c,Y)$, which we will sometimes emphasize with denoting it as $\mathbf{v}_{Y|c}$. Formally if $U_{p(c,Y)}$ the set of facts f that "match" (are instances of) $p(c,Y)$, then

$$\mathbf{v}_{Y|c}[d] = \Pr(f = p(c,d)|f \in U_{p(c,Y)}, \mathcal{T}, \mathcal{DB}, \Theta) \equiv \frac{1}{Z} w_Q(f = p(c,d))$$

Although here we only consider single-literal queries, we note that more complex queries can be answered by extending the theory: e.g., to find

$$\{\texttt{Y: uncle(joe,X),husband(X,Y)}\}$$

we could add the clause `q1(Y):-uncle(joe,X),husband(X,Y)` to the theory and find the answer to `q1(Y)`.

Since the goal of our reasoning system is to correctly answer queries using functions, we also introduce a notation for functions that answer particular types of queries: in particular, for a predicate symbol p, $f^p_{\texttt{io}}$ denotes a *query response function* for all queries with predicate p and mode `io`. We define a *query response function* for a query of the form $p(c,Y)$ to be a function which, when given a one-hot encoding of c, $f^p_{\texttt{io}}$ returns the appropriate conditional probability vector:

$$f^p_{\texttt{io}}(\mathbf{u}_c) \equiv \mathbf{v}_{Y|c} \tag{6}$$

We analogously define $f^p_{\texttt{oi}}$, Finally, we define $g^p_{\texttt{io}}$ to be the unnormalized version of this function, i.e., the weight of f according to $w_Q(f)$:

$$g^p_{\texttt{io}}(\mathbf{u}_c) \equiv w_Q(f)$$

For convenience, we will introduce another special DB predicate `any`, where `any`(a,b) is conceptually true for any pair of constants a,b; however, as we show below, the matrix

```
define compileMessage($L \rightarrow X$):
  assume wolg that $L = q(X)$ or $L = p(X_i, X_o)$
  generate a new variable name $\mathbf{v}_{L,X}$
  if $L = q(X)$ then
    emitOperation( $\mathbf{v}_{L,X} = \mathbf{v}_q$)
  else if $X$ is the output variable $X_o$ of $L$ then
    $\mathbf{v}_i$ = compileMessage($X_i \rightarrow L$)
    emitOperation( $\mathbf{v}_{L,X} = \mathbf{v}_i \cdot \mathbf{M}_p$ )
  else if $X$ is the input variable $X_i$ of $L$ then
    $\mathbf{v}_o$ = compileMessage($X_i \rightarrow L$)
    emitOperation( $\mathbf{v}_{L,X} = \mathbf{v}_o \cdot \mathbf{M}_p^T$ )
  return $\mathbf{v}_{L,X}$
```

```
define compileMessage($X \rightarrow L$):
  if $X$ is the input variable $X$ then
    return $\mathbf{u}_c$, the input
  else
    generate a new variable name $\mathbf{v}_X$
    assume $L_1, L_2, \ldots, L_k$ are the
      neighbors of $X$ excluding $L$
    for $i = 1, \ldots, k$ do
      $\mathbf{v}_i$ = compileMessage($L_i \rightarrow X$)
    emitOperation($\mathbf{v}_X = \mathbf{v}_1 \circ \cdots \circ \mathbf{v}_k$)
    return $\mathbf{v}_X$
```

Figure 5: Algorithm for unrolling belief propagation on a polytree into a sequence of message-computation operations. Notes: (1) if $L = p(X_o, X_i)$ then replace $\mathbf{M}_p$ with $\mathbf{M}_p^T$ (the transpose). (2) Here $\mathbf{v}_1 \circ \mathbf{v}_2$ denotes the Hadamard (componentwise) product, and if $k = 0$ an all-ones vector is returned.

$\mathbf{M}_{\texttt{any}}$ need not be explicitly stored. We also constrain clause heads to contain distinct variables which all appear also in the body.

3.2 Efficient inference for one-clause theories

We will start by considering a highly restricted class of theories $\mathcal{T}$, namely programs containing only one non-recursive polytree-limited clause r that obeys the restrictions above. We build a factor graph G_r for r as follows: for each logical variable W in the body, there is a random variable W; and for every literal $q(W_i, W_j)$ in the body of the clause, there is a factor with potentials $\mathbf{M}_q$ linking variables W_i and W_j. Finally, if the factor graph is disconnected, we add `any` factors between the components until it is connected. Figure 4 gives examples. The variables appearing in the clause's head are starred.

The correctness of this procedure follow immediately from the convergence of belief propagation on factor graphs for polytrees (Kschischang, Frey, & Loeliger, 2001).

BP over G_r can now be used to compute the conditional vectors $f^p_{\texttt{io}}(\mathbf{u}_c)$ and $f^p_{\texttt{oi}}(\mathbf{u}_c)$. For example to compute $f^p_{\texttt{io}}(\mathbf{u}_c)$ for clause 1, we would set the message for the evidence variable X to $\mathbf{u}_c$, run BP, and read out as the value of f the marginal distribution for Y.

3.3 Differentiable inference for one-clause theories

To make the final step toward integration of this algorithm with neural-network platforms, we must finally compute an explicit, differentiable, query response function, which computes $f^p_{\texttt{io}}(\mathbf{u}_c)$. To do this we "unroll" the message-passing steps into a series of operations. Figure 5 shows the algorithm used in the current implementation of TensorLog, which

Rule	r1: uncle(X,Y):- parent(X,W), brother(W,Y)	r2: uncle(X,Y):- aunt(X,W), husband(W,Y)	r3: status(X,T):- assign_tired(T), parent(X,W), infant(W),any(T,W)
Function	$g^{r1}_{\texttt{io}}(\vec{u}_c)$	$g^{r2}_{\texttt{io}}(\vec{u}_c)$	$g^{r3}_{\texttt{io}}(\vec{u}_c)$
Operation sequence defining function	$\mathbf{v}_{1,W} = \mathbf{u}_c\mathbf{M}_{\texttt{parent}}$ $\mathbf{v}_W = \mathbf{v}_{1,W}$ $\mathbf{v}_{2,Y} = \mathbf{v}_W\mathbf{M}_{\texttt{brother}}$ $\mathbf{v}_Y = \mathbf{v}_{2,Y}$	$\mathbf{v}_{1,W} = \mathbf{u}_c\mathbf{M}_{\texttt{aunt}}$ $\mathbf{v}_W = \mathbf{v}_{1,W}$ $\mathbf{v}_{2,Y} = \mathbf{v}_W\mathbf{M}_{\texttt{husband}}$ $\mathbf{v}_Y = \mathbf{v}_{2,Y}$	$\mathbf{v}_{2,W} = \mathbf{u}_c\mathbf{M}_{\texttt{parent}}$ $\mathbf{v}_{3,W} = \mathbf{v}_{\texttt{infant}}$ $\mathbf{W} = \mathbf{v}_{2,W} \circ \mathbf{v}_{3,W}$ $\mathbf{v}_{1,T} = \mathbf{v}_{\texttt{assign\_tired}}$ $\mathbf{v}_{4,T} = \mathbf{v}_W\mathbf{M}_{\texttt{any}}$ $\mathbf{T} = \mathbf{v}_{1,T} \circ \mathbf{v}_{4,T}$
Returns	$\mathbf{v}_Y$	$\mathbf{v}_Y$	$\mathbf{v}_T$

Table 1: Chains of messages constructed for the three sample clauses shown in Figure 4, written as functions in pseudo code.

follows previous work in translating belief propagation to differentiable form (Gormley, Dredze, & Eisner, 2015).

In the code, we found it convenient to extend the notion of input-output modes for a query, as follows: a variable X appearing in a literal $L = p(X, Y)$ in a clause body is an *nominal input* if it appears in the input position of the head, or any literal to the left of L in the body, and is an *nomimal output* otherwise. In Prolog a convention is that nominal inputs appear as the first argument of a predicate, and in TensorLog, if the user respects this convention, then "forward" message-passing steps use M_p rather than M_p^T (reducing the cost of transposing large $\mathcal{DB}$-derived matrices, since our message-passing schedule tries to maximize forward messages.) The code contains two mutually recursive routines, and is invoked by requesting a message from the output variable to a fictional output literal. The result will be to emit a series of operations, and return the name of a register that contains the unnormalized conditional probability vector for the output variable. For instance, for the sample clauses, the functions returned are shown in Table 1.

Here we use $g^r_{\texttt{io}}(\vec{u}_c)$ for the unnormalized version of the query response function build from G_r. One could normalize as follows:

$$f^p_{\texttt{io}}(\vec{u}_c) \equiv g^r_{\texttt{io}}(\vec{u}_c)/\|g^r_{\texttt{io}}(\vec{u}_c)\|_1 \tag{7}$$

where r is the one-clause theory defining p.

3.4 Multi-clause programs

We now extend this idea to theories with many clauses. We first note that if there are several clauses with the same predicate symbol in the head, we simply sum the unnormalized query response functions: e.g., for the predicate `uncle`, defined by rules r_1 and r_2, we would define

$$g^{\texttt{uncle}}_{\texttt{io}} = g^{r1}_{\texttt{io}} + g^{r2}_{\texttt{io}}$$

This is equivalent to building a new factor graph G, which would be approximately $\cup_i G_{ri}$, together global input and output variables, plus a factor that constrains the input variables of the G_{ri}'s to be equal, plus a factor that constrains the output variable of G to be the sum of the outputs of the G_{ri}'s.

A more complex situation is when the clauses for one predicate, p, use a second theory predicate q, in their body: for example, this would be the case if `aunt` was also defined in the theory, rather than the database. For a theory with no recursion, we can replace the message-passing operations $\mathbf{v}_Y = \mathbf{v}_X \mathbf{M}_q$ with the function call $\mathbf{v}_Y = g^q_{\texttt{io}}(\mathbf{v}_X)$, and likewise the operation $\mathbf{v}_Y = \mathbf{v}_X \mathbf{M}_q^T$ with the function call $\mathbf{v}_Y = g^q_{\texttt{oi}}(\mathbf{v}_X)$. It can be shown that this is equivalent to taking the factor graph for q and "splicing" it into the graph for p.

It is also possible to allow function calls to recurse to a fixed maximum depth: we must simply add an extra argument that tracks depth to the recursively-invoked g^q functions, and make sure that g^p returns an all-zeros vector (indicating no more proofs can be found) when the depth bound is exceeded. Currently this is implemented by marking learned functions g with the predicate q, a mode, and a depth argument d, and ensuring that function calls inside $g^p_{\texttt{io},d}$ to q always call the next-deeper version of the function for q, e.g., $g^q_{\texttt{io},d+1}$.

Computationally, the algorithm we describe is quite efficient. Assuming the matrices $\mathbf{M}_p$ exist, the additional memory needed for the factor-graph G_r is linear in the size of the clause r, and hence the compilation to response functions is linear in the theory size and the number of steps of BP. For ptree-SDKGs, G_r is a tree, the number of message-passing steps is also linear. Message size is (by design) limited to $|\mathcal{C}|$, and is often smaller in practice, due to sparsity or type restrictions (discussed below).

3.5 Implementation: TensorLog

Compilation and execution. The current implementation of TensorLog operates by first "unrolling" the belief-propagation inference to an intermediate form consisting of sequences of abstract operators, as suggested by the examples of Table 1. The "unrolling" code performs a number of optimizations to the sequence in-line: one important one is to use the fact that $\mathbf{v}_X \circ (\mathbf{v}_Y \mathbf{M}_{\texttt{any}}) = \mathbf{v}_X \|\mathbf{v}_Y\|_1$ to avoid explicitly building $\mathbf{M}_{\texttt{any}}$. These abstract operator sequences are then "cross-compiled" into expressions on one of two possible "back end" deep learning frameworks, Tensorflow (Abadi, Agarwal, Barham, Brevdo, Chen, Citro, Corrado, Davis, Dean, Devin, et al., 2016) and Theano (Bergstra, Breuleux, Bastien, Lamblin, Pascanu, Desjardins, Turian, Warde-Farley, & Bengio, 2010). The operator sequences can also be evaluated and differentiated on a "local infrastructure" which is implemented in the SciPy sparse-matrix package (Jones, Oliphant, & Peterson, 2014), which includes only the few operations actually needed for inference, and a simple gradient-descent optimizer.

The local infrastructure's main advantage is that it makes more use of sparse-matrix representations. In all the implementations, the matrices that correspond to KB relations are sparse. The messages corresponding to a one-hot variable binding, or the possible bindings to a variable, are sparse vectors in the local infrastructure, but dense vectors in the Tensorflow and Theano versions, to allow use of GPU implementations of multiplication of dense vectors and sparse matrices. (The implementation also supports grouping examples into minibatches, in which case the dense vectors become dense matrices with a number of rows equal to minibatch size.)

TensorLog compiles query response functions on demand, i.e., only as needed to answer queries or train. In TensorLog the parameters Θ are partitioned by the predicate they are associated with, making it possible to learn parameters for any selected subset of database predicates, while keeping the remainder fixed.

Typed predicates. One practically important extension to the language for the Tensorflow and Theano targets was include machinery for declaring types for the arguments of database predicates, and inferring these types for logic programs: for instance, for the sample program of Figure 1, one might include declarations like `actedIn(actor,film)` or `indicatesLabel(ngram,questionLabel)`. Typing reduces the size of the message vectors by a large constant factor, which increases the potential minibatch size and speeds up run-time by a similar factor.

Constraining the optimizer. TensorLog's learning changes the numeric score θ_f of every soft KG fact f using gradient descent. Under the proof-counting semantics used in TensorLog, a fact with a score of $\theta_f > 1$ could be semantically meaningful: for instance for $f =$ `costar(ginger_rogers,fred_astaire)` one might plausibly set θ_f to the number of movies those actors appeared in together. However it is not semantically meaningful to allow θ_f to be negative. To prevent this, before learning, for each KG parameter θ_f, we replace each occurrence of θ_f with $h(\tilde{\theta}_f)$ for the function $h = \ln(1 + e^x)$ (the "softplus" function), where $\tilde{\theta}_f \equiv h^{-1}(\theta_f)$. Unconstrained optimization is then performed to optimize the value of $\tilde{\theta}_f$ to some $\tilde{\theta}_f^*$ After learning, we update θ_f to be $h(\tilde{\theta}_f^*)$, which is always non-negative.

Regularization. By default, TensorLog trains to minimize unregularized cross-entropy loss. (Following common practice deep learning, the default loss function replaces the conventional normalizer of Equation 7 with a softmax normalization.) However, because modern deep-learning frameworks are quite powerful, it is relatively easy to use the cross-compiled functions produced by TensorLog in slight variants of this learning problem—often this requires only a few lines of code. For instance, Figure 6 illustrates how to add L1-regularization to TensorLog's loss function (and then train) using the Tensorflow backend.

Extension to multi-objective learning. It is also relatively easy to extend TensorLog in other ways. We will discuss several possible extensions which we have not, as yet, experimented with extensively, although we have verified that all can be implemented in the current framework.

For learning, TensorLog's training data consists of a set of queries $p(c_1, Y), \ldots, p(c_m, Y)$, and a corresponding set of desired outputs $\mathbf{v}_{Y|c_1}, \ldots, \mathbf{v}_{Y|c_m}$. It is possible to train with examples of multiple predicates: for instance, with the example program of Figure 1, one could include training examples for both `answer` and `matches`.

Alternative semantics for query responses. One natural extension would address a limitation of the SLP semantics, namely, that the weighting of answers relative to a query sometimes leads to a loss of information. For example, suppose the answers to `father(joe,Y)` are two facts `father(joe,harry)` and `father(joe,fred)`, each with weight 0.5. This answer does not distinguish between a world in which `joe`'s paternity is uncertain, and a world in which `joe` has two fathers. One possible solution is to learn parameters that set an appropriate soft threshold on each element of w_Q, e.g., to redefine f^p as

$$f^p(\mathbf{u})) = sigmoid(g^p(\mathbf{u}) + b^p)$$

where b^p is a bias term. The code required to do this for Tensorflow is below:

```
target = tlog.target_output_placeholder(function_spec)
```

```
tlog = tensorlog.simple.Compiler(db=”data.db”, prog=”rules.tlog”)
train_data = tlog.load_dataset(”train.exam”)
test_data = tlog.load_dataset(”test.exam”)
# data is stored dictionary mapping a function specification, like $p_{io}$,
# to a pair X, Y. The rows of X are possible inputs $f^p_{io}$, and the rows of
# Y are desired outputs.
function_spec = train_data.keys()[0]
# assume only one function spec
X,Y = train_data[function_spec]

# construct a tensorflow version of the loss function, and function used for inference
unregularized_loss = tlog.loss(function_spec)
f = tlog.inference(function_spec)
# add regularization terms to the loss
regularized_loss = unregularized_loss
for weight in tlog.trainable_db_variables(function_spec):
    regularized_loss = regularized_loss + tf.reduce_sum(tf.abs(weights))*0.01 # L1 penalty

# set up optimizer and inputs to the optimizer
optimizer = tf.train.AdagradOptimizer(rate)
train_step = optimizer.minimize(regularized_loss)
# inputs are a dictionary, with keys that name the appropriate variables used in the loss function
train_step_input = {}
train_step_input[tlog.input_placeholder_name(function_spec)] = X
train_step_input[tlog.target_output_placeholder_name(function_spec)] = Y

# run the optimizer for 10 epochs
session = tf.Session()
session.run(tf.global_variables_initializer())
for i in range(10):
    session.run(train_step, feed_dict=train_step_input)

# now run the learned function on some new data
result = session.run(f, feed_dict={tlog.input_placeholder_name(function_spec): X2})
```

Figure 6: Sample code for using TensorLog within Tensorflow. This code minimizes an alternative version of the loss function which includes and L1 penalty of the weights.

```
g = tlog.proof_count(function_spec) # g^p, computes w_Q
bias = tf.Variable(0.0, dtype=tf.float32, trainable=True)
f = tf.sigmoid(g + bias) # function used for inference
unregularized_loss = tf.nn.sigmoid_cross_entropy_with_logits(g+bias,target)
```

This extension illustrates an advantage of being able to embed TensorLog inferences in a deep network.

Extension to call out to the host infrastructure. A second extension is to allow TensorLog functions to "call out" to the backend language. Suppose, for example, we wish to replace the `classification` predicate in the example program of Figure 1 with a Tensorflow model, e.g., a multilayer perceptron, and that `buildMLP(q)` is function that constructs a an expression which evaluates the MLP on input `q`. We can instruct the compiler to include this model in place of the usual function $g^{\texttt{classification}}_{\texttt{io}}$ as follows:

```
plugins = tensorlog.program.Plugins()
plugins.define("classification/io", buildMLP)
tlog = simple.Compiler(db="data.db", prog="rules.tlog", plugins=plugins)
```

To date we have not experimentally explored this capability in depth; however, it would appear to be very useful to be able to write logical rules over arbitrary neurally-defined low-level predicates, rather than merely over KB facts. We note that the compilation approach also makes it easy to export a TensorLog predicate (e.g., the `answer` predicate defined by the logic) to a deep learner, as a function which maps a question to possible answers and their confidences. This might be useful in building a still more complex model non-logical model (e.g., a dialog agent which makes use of question-answering as a subroutine.)

4. Related Work

4.1 Hybrid logical/neural systems

There is a long tradition of embedding logical expressions in neural networks for the purpose of learning, but generally this is done indirectly, by conversion of the logic to a boolean formula, rather than developing a differentiable theorem-proving mechanism, as considered here. Embedding logic may lead to a useful architecture (Towell et al., 1990) or regularizer (Rocktschel et al., 2015; Hu et al., 2016).

More recently (Rocktäschel & Riedel, 2016) have proposed a differentiable theorem prover, in which a proof for an example is unrolled into a network. Their system includes representation-learning as a component, as well as a template-instantiation approach (similar to (Wang, Mazaitis, & Cohen, 2014)), allowing structure learning as well. However, published experiments with the system been limited to very small datasets. Another recent paper (Andreas, Rohrbach, Darrell, & Klein, 2016) describes a system in which non-logical but compositionally defined expressions are converted to neural components for question-answering tasks.

4.2 Explicitly grounded probabilistic first-order languages

Many first-order probabilistic models are implemented by "grounding", i.e., conversion to a more traditional representation. In the context of a deductive DB, a rule can be considered as a finite disjunction over ground instances: for instance, the rule

```
p(X,Y) :- q(Y,Z),r(Z,Y).
```

is equivalent to

$$\exists x \in \mathcal{C}, y \in \mathcal{C}, x \in \mathcal{C} : \texttt{p}(x,y) \vee \neg\texttt{q}(y,z) \vee \neg\texttt{r}(Z,Y)$$

For example, Markov logic networks (MLNs) are a widely-used probabilistic first-order model (Richardson & Domingos, 2006) in which a Bernoulli random variable is associated with each *potential* ground database fact (e.g., in the binary-predicate case, there would be a random variable for each possible $p(a,b)$ where a and b are any facts in the database and p is any binary predicate) and each ground instance of a clause is a factor. The Markov field built by an MLN is hence of size $O(|\mathcal{C}|^2)$ for binary predicates, which is much larger than the factor graphs used by TensorLog, which are of size linear in the size of the theory. In our experiments we compare to ProPPR, which has been elsewhere compared extensively to MLNs.

Inference on the Markov field can also be expensive, which motivated the development of probabilistic similarity logic (PSL), (Brocheler et al., 2010) a MLN variant which uses a more tractible hinge loss, as well as lifted relational neural networks (Sourek et al., 2015) and logic tensor networks (Serafini & Garcez, 2016) two recent models which grounds first-order theories to a neural network. However, any grounded model for a first-order theory can be very large, limiting the scalability of such techniques.

4.3 Stochastic logic programs and ProPPR

As noted above, TensorLog is very closely related to stochastic logic programs (SLPs) (Cussens, 2001). In an SLP, a probabilistic process is associated with a top-down theorem-prover: i.e., each clause r used in a derivation has an assocated probability θ_r. Let $N(r,E)$ be the number of times r was used in deriving the explanation E: then in SLPs, $\Pr_{\texttt{SLP}}(f) = \frac{1}{Z}\sum_{E \in \mathcal{E}x(f)} \prod_r \theta_r^{N(r,E)}$. The same probability distribution can be generated by TensorLog if (1) for each rule r, the body of r is prefixed with the literals `assign(RuleId,`r`),weighted(RuleId)`, where r is a unique identifier for the rule and (2) Θ is constructed so that $\theta_f = 1$ for ordinary database facts f, and $\theta_{\texttt{weighted(r)}} = \theta'_{\texttt{r}}$, where Θ' is the parameters for a SLP.

SLPs can be *normalized* or *unnormalized*; in normalized SLPs, Θ is defined so for each set of clauses S_p of clauses with the same predicate symbol p in the head, $\sum_{r \in S_p} \theta_r = 1$. TensorLog can represent both normalized and unnormalized SLPs (although clearly learning must be appropriately constrained to learn parameters for normalized SLPs.) Normalized SLPs generalize probabilistic context-free grammars, and unnormalized SLPs can express Bayesian networks or Markov random fields (Cussens, 2001).

ProPPR (Wang et al., 2013) is a variant of SLPs in which (1) the stochastic proof-generation process is augmented with a reset, and (2) the transitional probabilities are based on a normalized soft-thresholded linear weighting of features. The first extension to SLPs can be easily modeled in TensorLog, but the second cannot: the equivalent of

ProPPR's clause-specific features can be incorporated, but they are globally normalized, not locally normalized as in ProPPR.

ProPPR also includes an approximate grounding procedure which generates networks of bounded size. Asymptotic analysis suggests that ProPPR should be faster for very large database and small numbers of training examples (assuming moderate values of ϵ and α are feasible to use), but that TensorLog should be faster with large numbers of training examples and moderate-sized databases.

5. Experiments

5.1 Inference tasks

We compared TensorLog's inference time (using the local infrastructure) with ProbLog2, a mature probabilistic logic programming system which implements the tuple independence semantics, on two inference problems described in (Fierens et al., 2016). One is a version of the "friends and smokers" problem, a simplified model of social influence. In (Fierens et al., 2016) small graphs were artificially generated using a preferential attachment model, the details of which were not described; instead we used a small existing network dataset[2] which displays preferential-attachment statistics. The inference times we report are for the same inference tasks, for a subset of 120 randomly-selected entities. As shown in Table 2, in spite of querying six times as many entities, TensorLog is many times faster.

We also compare on a path-finding task from (Fierens et al., 2016), which is intended to test performance on deeply recursive tasks. The goal here is to compute fixed-depth transitive closure on a grid: in (Fierens et al., 2016) a 16-by-16 grid was used, with a maximum path length of 10. Again TensorLog shows much faster performance, and better scalability, as shown in Table 3 by run times on a larger 64-by-64 grid. We set TensorLog's maximum path length to 99 for the larger grid.

5.2 Learning Tasks

We also compared experimentally with ProPPR on several standard benchmark learning tasks. We chose two traditional relational learning tasks on which ProPPR outperformed plausible competitors, such as MLNs. One was the CORA citation-matching task (from (Wang et al., 2013)) with hand-constructed rules.[3]. A second was learning the most common relation, "affects", from UMLS, using a rule set learned by the algorithm of (Wang et al., 2014). Finally, motivated by recent comparisons between ProPPR and embedding-based approaches to knowledge-base completion (Wang & Cohen, 2016), we also compared to ProPPR on two relation-prediction tasks involving WordNet, again using rules from the (non-recursive) theories used in (Wang & Cohen, 2016).

In all of these tasks parameters are learned on a separate training set. For TensorLog's learner, we used the local infrastructure with the default loss function (unregularized cross-

2. The Citeseer dataset from (Lin & Cohen, 2010).
3. We replicated the experiments with the most recent version of ProPPR, obtaining a result slightly higher than the 2013 version's published AUC of 80.0

	Social Influence Task	
ProbLog2	20 nodes	40-50 sec
TensorLog	3327 nodes	**9.2 msec**

Table 2: Comparison to ProbLog2 on the "friends and smokers" inference task.

	Path-finding		
	Size	Time	Acc
ProbLog2	16x16 grid, $d = 10$	100-120 sec	
TensorLog	16x16 grid, $d = 10$	**2.1 msec**	
	64x64 grid, $d = 99$	2.2 msec	
trained	16x16 grid, $d = 10$	6.2 msec	99.89%

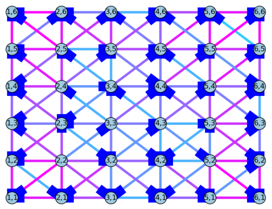

Table 3: Comparison to ProbLog2 on path-finding in a grid.

Grid Size	Max Depth	# Graph Nodes		Acc		Time (30 epochs)	
		Local	TF	Local	TF	Local	TF
16	10	68	2696	**99.9**	97.2	37.6 sec	**1.1 sec**
18	12	80	3164	93.9	**96.9**	126.1 sec	**1.8 sec**
20	14	92	3632	25.2	**99.1**	144.9 sec	**2.8 sec**
22	16	104	4100	8.6	**98.4**	83.8 sec	**4.2 sec**
24	18	116	4568	**2.4**	0.0	611.7 sec	**6.3 sec**

Table 4: Learning for the path-finding task with local and Tensorflow (TF) backends.

	ProPPR	TensorLog
CORA (13k facts,10 rules)	AUC 83.2	AUC **97.6**
UMLS (5k facts, 226 rules)	acc 49.8	acc **52.5**
Wordnet (276k facts)		
Hypernym (46 rules)	acc **93.4**	acc 93.3
Hyponym (46 rules)	acc 92.1	acc **92.8**

Table 5: Comparison to ProPPR on relational learning tasks.

entropy loss), using a fixed-rate gradient descent learner with the learning rate to 0.1, and 30 epochs.[4] We also used the default parameters for ProPPR's learning.

Table 5 shows that the accuracy of the two systems after learning is quite comparable, even with a rather simplistic learning scheme. ProPPR, of course, is not well suited to tight integration with deep learners.

5.3 Path-finding after learning

The results of Section 5.1 demonstrate that TensorLog's approximation to ProbLog2's semantics is efficient, but not that it is useful. To demonstrate that TensorLog can efficiently and usefully approximate deeply recursive concepts, we posed a learning task on the 16-by-16 grid, with a maximum depth of 10, and trained TensorLog to approximate the distribution for this task. The dataset consists of 256 grid cells connected by 2116 edges, so there are 256 example queries of the form `path(a,X)` where a is a particular grid cell. We picked 1/3 of these queries as test, and the remainder as train, and trained so that that the single positive answer to the query `path(a,X)` is the extreme corner closest to `a`—i.e., one of the corners (1,1), (1,16), (16,1) or (16,16). We set the initial weights of the edges uniformly to 0.2.

Training for 30 epochs with the local backend and a fixed-rate gradient descent learner, using a learning rate of 0.01, brings the accuracy from 0% to 99.89% for test cases (averaged over 10 trials, with different train/test splits). Learning takes less than 1.5 sec/epoch. After learning query times are still quite fast, as shown in the table.

The table also includes a visualization of the learned weights for a small 6x6 grid. For every pair of adjacent grid cells u, v, there are two weights to learn, one for the edge from u to v and one for its converse. For each weight pair, we show a single directed edge (the heavy blue squares are the arrows) colored by the magnitude of the difference.

We observe that ProbLog2, in addition to implementing the full tuple-independence semantics, implements a much more expressive logic than considered here, including a large portion of full Prolog, while in contrast TensorLog includes only a subset of Datalog. So to some extent this comparison is unfair.

We also observe that although this task seems simple, it is quite difficult for probabilistic logics, because of deeply recursive theories lead to large, deep proofs. While TensorLog's inference schemes scale well on this task, is still challenging to optimize the parameters, especially for larger grid sizes. One problem is that unrolling the inference leads to very large graphs, especially after they are compiled to the relatively fine-grained operations used in deep-learning infrastructure. Table 4 shows the size of the networks after compilation to Tensorflow for various extensions of the 16-by-16 depth 10 task. Although growth is linear in depth, the constants are large: e.g., the Tensorflow 64-by-64 depth 99 network does not fit in memory for a 4Gb GPU.

A second problem is that the constructed networks are very deep, which leads to problems in optimization. For the smaller task, the local optimizer (which is a fixed-rate gradient descent method) required careful tuning of the initial weights and learning rate to reliably converge.

4. Thirty epochs approximately matches ProPPR's runtime on a single-threaded machine.

Original KB		Extended KB		Num Examples		
Num Tuples	Num Relations	Num Tuples	Num Relations	Train	Devel	Test
421,243	10	1,362,670	12	96,182	20,000	10,000

Table 6: Statistics concerning the WikiMovies dataset.

Method	Accuracy	Time per epoch
Subgraph/question embedding	93.5%	
Key-value memory network	93.9%	
TensorLog (1,000 training examples)	89.4%	6.1 sec
TensorLog (10,000 training examples)	94.8%	1.7 min
TensorLog (96,182 training examples)	**95.0%**	49.5 min

Table 7: Experiments with the WikiMovies dataset. The first two results are taken from (Miller et al., 2016).

The size and complexity of this task suggested a second set of experiments, where we varied the task complexity, while fixing the parameters of two optimizers. For the local optimizer we fixed the parameters to those used the 16-by-16 depth 10 task, and for the Tensorflow backend, we used a the `AdagradOptimizer` with a default learning rate of 1.0, running for 30 epochs. The results are shown in Table 4 (averaged over 10 trials for each datapoint), and they illustrate several of the advantages of using a mature deep-learning framework as the backend of TensorLog.

- In general learning is many times faster for the Tensorflow backend, which uses a GPU processor, than using the local infrastructure.[5]
- Although they do not completely eliminate the need for hyperparameter tuning, the more sophisticated optimizers available in Tensorflow do appear to be more robust. In particular, Adagrad performs well up to a depth of around 16, while the fixed-rate optimizer performs well only for depths 10 and 12.

We conjecture that good performance on larger grid sizes would require use of gradient clipping.

5.4 Answering Natural-Language Questions Against a KB

As larger scale experiment, we used the WikiMovies question-answering task proposed by (Miller et al., 2016). This task is similar to the one shown in Figure 1. The KB consists of over 420k tuples containing information about 10 relations and 16k movies. Some sample questions with their answers are below, with double quotes identifying KB entities.

5. Learning times for the local infrastructure are quite variable for the larger sizes, because numerical instabilities often cause the optimizer to fail. In computing times we discard runs where there is overflow but not when there is underflow, which is harder to detect. The high variance accounts for the anomolously low average time for grid size 22.

- Question: Who acted in the movie Wise Guys?
  Answers: "Harvey Keitel", "Danny DeVito", "Joe Piscopo", ...

- Question: what is a film written by Luke Ricci?
  Answer: "How to be a Serial Killer"

We encoded the questions into the KB by extending it with two additional relations: `mentionsEntity(Q,E)`, which is true if question `Q` mentions entity `E`, and `hasFeature(Q,W)`, which is true if question `Q` contains feature `W`. The entities mentioned in a question were extracted by looking for every longest match to a name in the KB. The features of a question are simply the words in the question (minus a short stoplist).

The theory is a variant of the one given as an example in Figure 1. The main difference is that because the simple longest-exact-match heuristic described above identifies entities accurately for this dataset, we made `mentionsEntity` a hard KB predicate. We also extended the theory to handle questions with answers that are either movie-related entities (like the actors in the first example question) or movies (as in the second example. Finally, we simplified the question-classification step slightly. The final theory contains two rules and two "soft" unary relations $\texttt{QuestionType}_{R,1}$, $\texttt{indicatesQuestionType}_{R,2}$ for each relation R in the original movie KB. For example, for the relation `directedBy` the theory has the two rules

```
answer(Question,Movie) :-
    mentionsEntity(Question,Entity), directedBy(Movie,Entity),
    hasFeature(Question,Word), indicatesQuestionType_directedBy,1(Word)
answer(Question,Entity) :-
    mentionsEntity(Question,Movie), directedBy(Movie,Entity),
    hasFeature(Question,Word), indicatesQuestionType_directedBy,2(Word)
```

The last line of each rule acts as a linear classifier for that rule.

For efficiency we used three distinct types of entities (question ids, entities from the original KB, and word features) and the Tensorflow backend, with minibatches of size 100 and an Adagrad optimizer with a learning rate of 0.1, running for 20 epochs, and no regularization. We compare accuracy results with two prior neural-network based methods which have been applied to this task. As shown in Table 7, TensorLog performs better than the prior state-of-the-art on this task, and is quite efficient.

6. Concluding Remarks

In this paper, we described a scheme to integrate probabilistic logical reasoning with the powerful infrastructure that has been developed for deep learning. The end goal is to enable deep learners to incorporate first-order probabilistic KBs, and conversely, to enable probabilistic reasoning over the outputs of deep learners. TensorLog, the system we describe here, makes this possible to do at reasonable scale using conventional neural-network platforms.

This paper contains several interrelated technical contributions. First, we identified a family of probabilistic deductive databases (PrDDBs) called polytree-limited stochastic deductive knowledge graphs (ptree-SDKGs) which are tractable, but still reasonably expressive. This language is a variant of SLPs, and it is maximally expressive, in that one cannot drop the polytree restriction, or switch to a possible-worlds semantics, without making inference intractible. We argue above that logics which are not tractable (i.e., are #P or worse in complexity) are unlikely to be practically incorporated into neural networks.

Second, we presented an algorithm for performing inference for ptree-SDKGs, based on belief propagation. Computationally, the algorithm is quite efficient. Assuming the matrices $\mathbf{M}_p$ exist, the additional memory needed for the factor-graph G_r is linear in the size of the clause r, and hence the compilation is linear in the theory size and recursion depth. To our knowledge use of BP for first-order inference in this setting is novel.

Finally, we present an implementation of this logic, called TensorLog. The implementation makes it possible to both call TensorLog inference within neural models, or conversely, to call neural models within TensorLog.

The current implementation of TensorLog includes a number of restrictions. Two backends are implemented, one for Tensorflow and one for Theano, but the Tensorflow backend has been more extensively tested and evaluated. We are also exploring compilation to PyTorch[6], which supports dynamic networks. We also plan to implement support for more stable optimization (e.g., gradient clipping), and better support for debugging.

As noted above, TensorLog also makes it possible to replace components of the logic program (e.g., the `classification` or `matches` predicate) with submodels learned in the deep-learning infrastructure. Alternatively, one can export a `answer` predicate defined by the logic to a deep learner, as a function which maps a question to possible answers and their confidences; this might be useful in building a still more complex model non-logical model (e.g., a dialog agent which makes use of question-answering as a subroutine.) In future work we hope to explore these capabilities.

We also note that although the experiments in this paper assume that theories are given, the problem of learning programs in TensorLog is also of great interest. Some early results from the authors on this problem are discussed elsewhere (Yang, Yang, & Cohen, 2017).

Acknowledgments

Thanks to William Wang for providing some of the datasets used here; and to William Wang and many other colleagues contributed with technical discussions and advice. The author is greatful to Google for financial support, and also to NSF for their support of his work via grants CCF-1414030 and IIS-1250956.

6. pytorch.org

Appendix A. Proofs

Theorem 1 *Computing $P(f|Q)$ (relative to a SDKG $\mathcal{T},\mathcal{DB}$) for all possible answers f of the query Q is #P-hard, even if there are only two such answers, the theory contains only two non-recursive clauses, and the KG contains only 13 facts.*

We will reduce counting proofs for 2PSAT to computing probabilities for SDKGs. 2PSAT is a #P-hard task where the goal is to count the number of satisfying assignments to a CNF formula with only two literals per clause, all of which are positive (Suciu et al., 2011). Hence a 2PSAT formula is of the form

$$(x_{a_1} \vee x_{b_1}) \wedge \ldots \wedge (x_{a_n} \vee \ell_{b_n})$$

where the variables are all binary variables x_i from $X = \{x_1, \ldots, x_n\}$, and each a_i and b_i is a index between 1 and n. The subformula $(x_{a_i} \vee x_{b_i})$ is called the i-th clause below.

The database contains the two facts `assign_yes(yes)` and `assign_no(no)` with weight 1, and two facts `binary(0)` and `binary(1)` with weights 0.5. It also contains a definition of the predicate `either_of`, containing the following three weight 1 facts: `either_of(0,1)`, `either_of(1,0)`, and `either_of(1,1)`. There are a total of 7 facts in the database.

```
sat(Y) :-
      assign_yes(Y), binary(X_1), ..., binary(X_n),
      either_of(X_{a_1},X_{b_1}),
      ...,
      either_of(X_{a_n},X_{b_n}),
sat(Y) :-
      assign_no(Y).
```

In the first line of the first rule, an assignment to the x_i's is selected, with uniform probability. It is easy to see that the literal `either_of(X`$_{a_i}$`,X`$_{b_i}$`)` will succeed iff the i-th clause is made true by this assignment. Hence the first rule of the theory will succeed exactly k times, where k is the number of satisfying assignments for the formula. The second clause succeeds once, so

$$p = \Pr(\texttt{sat(yes)}|\texttt{sat(Y)}) = \frac{k}{k+1}$$

If p could be computed efficiently, one could solve the equation above for k and use the result to determine the number of satisfying assignments to the 2PSAT formula.

Theorem 3 *Computing $P(f)$ in the tuple-independent possible-worlds semantics for a single ground fact f is #P-hard.*

We again reduce counting assignments for 2NSAT to computation of $p = \Pr(\texttt{sat(yes)})$. In this case the DB contains n facts of the form `x`$_1$`(1)`, `x`$_2$`(1)`, ..., `x`$_n$`(1)`, all with weights 0.5, and the additional fact `assign_yes(yes)`.

We can now encode the 2PSAT formula with the following theory. For each clause i let $j1$ and $j2$ be the indices of the two literals in that clause. We construct two theory rules for each clause i:

```
sat$_i$(Y) :- x$_{j1}$(Y).
sat$_i$(Y) :- x$_{j2}$(Y).
```

Finally we add a binary tree of $O(\log(n))$ rules, each of which test success of two other subpredicates, and the last of which tests succeeds only of all the clause$_i$ predicates succeed. For instance, for $n = 8$, we would define sat(Y) as

```
sat$_{1:2}$(Y) :- clause$_1$(Y),clause$_2$(Y).
sat$_{2:3}$(Y) :- clause$_2$(Y),clause$_3$(Y).
sat$_{1:4}$(Y) :- sat$_{1:2}$(Y),sat$_{2:3}$(Y).
sat$_{5:6}$(Y) :- clause$_5$(Y),clause$_6$(Y).
sat$_{7:8}$(Y) :- clause$_7$(Y),clause$_8$(Y).
sat$_{5:8}$(Y) :- sat$_{5:6}$(Y),sat$_{7:8}$(Y).
sat(Y) :- sat$_{1:4}$(Y),sat$_{5:8}$(Y),assign_yes(Y).
```

Note that for each variable x_j, an I drawn from the database distribution may contain either x$_j$(1) or not, so there are 2^n possible interpretations. Each of these corresponds to a boolean assignment, where $x_j = 1$ in the assignment exactly when x$_j$(1) is in the corresponding interpretation. Clearly the clause$_i$ predicate succeeds exactly when the i-th clause is satisfied, and hence $p = \Pr(\texttt{sat(yes)}|\mathcal{DB}, |T)$ is thus exactly $\frac{k}{2^n}$, where k is the number of satisfying assignments.

This theory is quite simple: it contains no binary predicates, and (if one tests success of all clause predicates in a tree) the rules are all very short. It is thus difficult to identify syntactic restrictions which might make proof-counting tractible for the possible-worlds scenario.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Andreas, J., Rohrbach, M., Darrell, T., & Klein, D. (2016). Learning to compose neural networks for question answering. *CoRR*, *abs/1601.01705*.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., & Bengio, Y. (2010). Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pp. 1–7.

Brocheler, M., Mihalkova, L., & Getoor, L. (2010). Probabilistic similarity logic. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

Cussens, J. (2001). Parameter estimation in stochastic logic programs. *Machine Learning*, *44*(3), 245–271.

De Raedt, L., & Kersting, K. (2008). *Probabilistic inductive logic programming*. Springer.

Fierens, D., Broeck, G. V. D., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., & Raedt, L. D. (2016). Inference and learning in probabilistic logic programs using weighted boolean formulas. To appear in Theory and Practice of Logic Programming.

Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., & Tenenbaum, J. B. (2012). Church: a language for generative models. *arXiv preprint arXiv:1206.3255*.

Gormley, M. R., Dredze, M., & Eisner, J. (2015). Approximation-aware dependency parsing by belief propagation. *Transactions of the Association for Computational Linguistics (TACL)*.

Hu, Z., Ma, X., Liu, Z., Hovy, E., & Xing, E. (2016). Harnessing deep neural networks with logic rules. *arXiv preprint arXiv:1603.06318*.

Jones, E., Oliphant, T., & Peterson, P. (2014). *{SciPy}: open source scientific tools for {Python}*.

Kimmig, A., Mihalkova, L., & Getoor, L. (2015). Lifted graphical models: a survey. *Machine Learning*, *99*(1), 1–45.

Kschischang, F. R., Frey, B. J., & Loeliger, H.-A. (2001). Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, *47*(2), 498–519.

Lin, F., & Cohen, W. W. (2010). Semi-supervised classification of network data using very few labels. In Memon, N., & Alhajj, R. (Eds.), *ASONAM*, pp. 192–199. IEEE Computer Society.

Miller, A., Fisch, A., Dodge, J., Karimi, A.-H., Bordes, A., & Weston, J. (2016). Key-value memory networks for directly reading documents. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1400–1409, Austin, Texas. Association for Computational Linguistics.

Muggleton, S., et al. (1996). Stochastic logic programs. *Advances in inductive logic programming*, *32*, 254–264.

Poole, D. (1997). The independent choice logic for modelling multiple agents under uncertainty. *Artificial intelligence*, *94*(1), 7–56.

Ramakrishnan, R., & Ullman, J. D. (1995). A survey of deductive database systems. *The journal of logic programming*, *23*(2), 125–149.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Mach. Learn.*, *62*(1-2), 107–136.

Rocktäschel, T., & Riedel, S. (2016). Learning knowledge base inference with neural theorem provers. In *NAACL Workshop on Automated Knowledge Base Construction (AKBC)*.

Rocktschel, T., Singh, S., & Riedel, S. (2015). Injecting logical background knowledge into embeddings for relation extraction. In *Proc. of ACL/HLT*.

Serafini, L., & Garcez, A. d. (2016). Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *arXiv preprint arXiv:1606.04422*.

Sourek, G., Aschenbrenner, V., Zelezný, F., & Kuzelka, O. (2015). Lifted relational neural networks. *CoRR*, *abs/1508.05128*.

Suciu, D., Olteanu, D., Ré, C., & Koch, C. (2011). Probabilistic databases. *Synthesis Lectures on Data Management*, *3*(2), 1–180.

Towell, G., Shavlik, J., & Noordewier, M. (1990). Refinement of approximate domain theories by knowledge-based artificial neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, Massachusetts. MIT Press.

Wang, W. Y., & Cohen, W. W. (2016). Learning first-order logic embeddings via matrix factorization. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, New York, NY. AAAI.

Wang, W. Y., Mazaitis, K., & Cohen, W. W. (2013). Programming with personalized PageRank: a locally groundable first-order probabilistic logic. In *Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management*, pp. 2129–2138. ACM.

Wang, W. Y., Mazaitis, K., & Cohen, W. W. (2014). Structure learning via parameter learning. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pp. 1199–1208. ACM.

Yang, F., Yang, Z., & Cohen, W. W. (2017). Differentiable learning of logical rules for knowledge base completion. *arXiv preprint arXiv:1702.08367*.